Lab 2: pekare och funktioner

Imperativ Programmering DT501G, HT 2019
6 december 2019

Deadline

Deadline för inlämning är **6 december**.

Lärandemål

- Funktionsanrop i C.
- Felsökning (debugger).
- Inargument till ett program via `argv` och `argc`.
- Förstå skillnaden på värdeanrop och pekaranrop, och kunna använda dem på rätt sätt.
- Kunna hantera **`const`** för pekare och andra variabler.
- Definiera egna typer med **`typedef`**.
- Kunna använda sammansatta datatyper, **`struct`**.
- Kunna hantera och traversera strängar och fält.
- Förstå hur tal representeras i en binär dator.
- Kunna definiera och använda testfall.

Frågor

Frågorna som behöver besvaras är markerade i marginalen. Skriv en textfil (rapport.txt) där du svarar på frågorna, och lägg till den i ditt git-repo, tillsammans med källkoden.

Om testfall och felhantering

För alla uppgifter här gäller att du måste kunna hantera, till viss del, felaktig inmatning från användaren. Framför allt behöver du se till att du får rätt antal argument vid inmatning, och att programmet inte kraschar om det kommer något oväntat. Se till att hantera det på ett vettigt sätt (till exempel genom att skriva ut ett felmeddelande, och avsluta programmet med en felkod). Däremot måste du inte hantera om användaren matar in bokstäver i stället för siffror eller vice versa.

Glöm inte det här!

För varje uppgift, utom 2.9–2.12, specificera minst två testfall som du har använt för att kontrollera att programmet gör som det ska. De båda testfallen ska testa olika saker (inte bara två varianter av giltigt input, t ex). Ange dina testfall, och förväntat resultat, som kommentarer i din källkod.

Exempel för uppgift 2.1:

```
//...
int main( int argc, char *argv[] )
{
  // ...
}
```

```
/* TESTFALL
Test med heltal och flyttal
input: task_1 3 2.5
resultat: 5.5

Test med negativa tal
input: task_1 -8 -5
resultat: -13

Test med ogiltig input
input: task_1 0
resultat: Programmet avslutas med ett felmeddelande som säger att
mer input behövs.
*/
```

Dokumentation

Alla funktioner (utom main) ska dokumenteras enligt de givna instruktionerna (se "Generella instruktioner"). Dokumentera också större kodblock så att det framgår vad det är tänkt att de ska göra.

Uppgifter

2.1 En funktion

Skriv ett program med en funktion `double sum( double a, double b )` som returnerar summan av talen `a` och `b`. Ditt program ska läsa in två tal från kommandoraden (via `argv`) och skriva ut summan. Du kan använda funktionen `atof`[1] från `stdlib.h` för att konvertera från en sträng från `argv` till ett flyttal.

```
./task_1 3 -2.5
0.500000
```

2.2 En funktion med pekare

Skriv ett program med en funktion `sum` som, i stället för att *returnera* summan, skriver det till en av parametrarna. För att det här ska fungera behöver alltså (åtminstone) en av parametrarna vara en pekare. (Är din/dina pekare `const` eller inte? Varför? Motivera för dig själv du bör göra.)

Från användarens perspektiv ska det här programmet vara identiskt med det i den förra uppgiften.

2.3 Fält

Skriv ett program som läser in 3 numeriska värden från kommandoraden, lagrar dem internt i ett `double`-fält som rymmer 3 värden, och skriv sedan ut värdena baklänges, med blanktecken emellan.

Exempel:

```
./task_3 1 3 5.6
5.600000 3.000000 1.000000
```

[1] Se http://www.fortran-2000.com/ArnaudRecipes/Cstd/2.13.html#atof

2.4 Vektorlängd

Skriv ett program som läser in 3 numeriska värden från kommandoraden, och lagrar dem internt i ett **`double`**-fält som rymmer 3 värden Skriv också en funktion som beräknar längden på en 3-vektor, det vill säga $l = \sqrt{x^2 + y^2 + z^2}$. Låt ditt program skriva ut längden av den inmatade vektorn.

Du kan antingen låta din funktion ta tre flyttal som parametrar, eller ett fält. (Fundera på om det finns någon praktisk skillnad mellan de två alternativen. Är det ena mer effektivt än det andra, när det gäller minnesanvändning? *Tips:* kolla inne i din funktion hur många bytes varje **`double`** använder med hjälp av **`sizeof(double)`**, jämfört med en pekare **`sizeof(double*)`**.)

```
./task_4 1 3 5.6
6.431174
```

2.5 Vektorprodukt

Skriv nu ett program med en funktion som beräknar vektorprodukten (kryssprodukten) av två vektorer, och skriver ut resultatet. Vektorprodukten för två vektorer (x_1, y_1, z_1), (x_2, y_2, z_2) beräknas så här:

$$\begin{aligned} x &= y_1 z_2 - z_1 y_2 \\ y &= z_1 x_2 - x_1 z_2 \\ z &= x_1 y_2 - y_1 x_2 \end{aligned}$$

Din funktion ska ta *tre fält som argument*: ett för varje vektor.

I det här fallet är det svårt att *returnera* resultatet från funktionen som beräknar produkten, eftersom det bara går att returnera *ett* värde i C (inga tupler, och inga fält). Alltså måste du använda ett *ut-argument*, precis som du gjorde i uppgift 2.2.

I din funktion, var noga med att ange **`const`** på ett korrekt sätt för dina parametrar. Pekarargument där inget ändras ska generellt sett deklareras med **`const`**.

Exemplet nedan visar hur det ska se ut. Programmet läser alltså in två vektorer $\vec{u} = (1, 2, 3)$ och $\vec{v} = (4, 5, 6)$ och skriver ut $\vec{u} \times \vec{v}$.

```
./task_5 1 2 3 4 5 6
-3.0 6.0 -3.0
```

I den här deluppgiften, och endast här, ska du *inte* kolla i ditt program att du får in tillräckligt antal argument i `argv`. Kör nu ditt program med bara tre inputvärden. Då kommer det antagligen att krascha. Stega igenom programmet med en avlusare (använd t ex *debug* i Geany). På vilken rad kraschar programmet? Och varför?

Fråga 1

För att kunna stega igenom programmet ska du kompilera det med två extra flaggor till gcc: dels `-g`, som ser till att avlusningsinformation finns med i programmet, och dels `-O0` (alltså "O noll"), som ser till att inga optimeringar görs, som skulle kunna göra att det kompilerade programmet inte gör riktigt det det ser ut som utifrån källkoden. Med andra ord: `gcc task_5.c -Wall -std=c99 -g -O0`.

2.6 Rimstuga

Skriv ett program som tar två strängar som argument och avgör om de rimmar eller ej. För den här uppgiften använder vi en väldigt lös definition på att rimma: det räcker om de tre sista bokstäverna är samma.

Även i den här uppgiften ska själva funktionaliteten, att avgöra om två strängar rimmar eller ej, ligga i en egen funktion, och inte i `main`.

Ditt program ska dels skriva ut resultatet på skärmen (”ja” eller ”nej”), och dels *returnera* ett värde. I vanliga fall returnerar `main` 0, såvida inget gick fel. Returnera 0 om inargumenten rimmar, eller 1 om de inte rimmar, eller en annan felkod om något gick fel (t ex om användaren matade in för få ord, eller om de inte innehåller minst tre bokstäver). Använd gärna `strlen` från biblioteket `string.h` för att kolla längden på strängar.

Exempel:

```
./task_6 imperativ programmering
nej

./task_6 programmering schmogrammering
ja
```

2.7 Kopiera strängar

Funktionen `strcpy`, från standardbiblioteket `string.h`, kopierar en sträng (inklusive slutttecknet) till en annan. Den här uppgiften går ut på att skriva en egen `strcpy`-funktion, utan att använda standardfunktionen.[2]

Skriv en funktion som kopierar en sträng enligt följande mall.

```
/*
 * Kopiera en sträng till en annan. Inkluderar slutttecknet, och
 * avslutar kopieringen där.
 *
 * Paramterar:
 * destination: Pekar på strängen där resultatet ska hamna.
 * source: Pekar på input-strängen.
 *
 * Returnerar:
 * Antalet tecken som har kopierats (inklusive
 * avslutningstecknet).
 */
int string_copy( char *destination, const char *source )
{
  /*...*/
}
```

(Kanske kan det tyckas bakvänt att ha destinationssträngen till vänster, men det följer ju samma form som vanlig tilldelning, och även `strcpy`. Notera att returvärdet är en **`int`** för den här funktionen, och inte en **`char`***, som i standardfunktionen.)

Använd din funktion i ett program som läser in två strängar med `scanf`, kopierar den ena till den andra, och skriver ut dem igen.

```
./task_7
hejsan
svejsan
hejsan hejsan
```

Kopiering av strängar är en notorisk felkälla (och något som också har gett upphov till flera ”exploits”[3]). Vad skulle hända när din funktion körs i följande fall?

Fråga 2
- Om fältet som `source` pekar på är större än fältet som `destination` pekar på?

Fråga 3
- Om fältet som `source` pekar på är mindre än fältet som `destination` pekar på?

[2] http://www.fortran-2000.com/ArnaudRecipes/Cstd/2.14.html#strcpy

[3] Se t ex den här länken och den här om du är intresserad.

Fråga 4
- Om `source` och `destination` pekar på samma adress?

Fråga 5
- Om det inte finns något avslutande 0-tecken i fältet som `source` pekar på?

Du behöver inte implementera någon avancerad felkontroll för att hantera alla de här fallen, men fundera på vad som kan hända. I ditt program ser du kanske också till att mata in strängar som är giltiga så att problematiska fall inte uppstår, men tänk dig i dina svar på vad som skulle kunna hända om någon använder din funktion med andra **`char`**$*$.

2.8 En datastruktur för bilder

De nästföljande uppgifterna handlar om bildmanipulation. Börja med att skapa en **`struct`** som innehåller den information som behövs om en bild. Vi kommer att använda gråskalebilder, så det som är användbart är dels ett fält med pixelvärden (hur ljus varje pixel är), dels data om hur hög och bred bilden är, och dels data som anger maxvärdet på en pixel (alltså, vilket värde motsvarar helvitt). Gör alltså en **`struct`** som innehåller två heltal (vilken heltalstyp är lämpligast här?) och ett fält med tal. Eftersom storleken på fält måste definieras redan när programmet kompileras[4] så får du se till att ta i tillräckligt för att ”alla” bilder ska få plats.Men du kan utgå från att du inte kommer att hantera några bilder som är större än 500×500 pixlar. Man kan tänka sig antingen ett vanligt endimensionellt fält för att lagra pixlarna (t ex **`int[]`**), eller ett tvådimensionellt fält (t ex **`int[][]`**). (Vilket föredrar du? Varför?)

Gör också en **`typedef`** så att du enkelt kan skapa variabler av den här **`struct`**-typen i ditt program. Alltså:

```
typedef struct {
  // fyll i här
} image;
```

Den här datatypen **`typedef struct`** `image` kommer du sedan att använda i alla de kommande uppgifterna.

Innan du faktiskt ska läsa in en bild och *göra* något så ska du kontrollera vad som händer om du skickar din **`struct`** som en kopia eller via en pekare.

Skapa två funktioner:

```
void image_thrash_by_value( image I, unsigned int c )
{
  printf( "%u\n", c );
  image_thrash_by_value( I, c+1 );
}
void image_thrash_by_pointer( const image * I, unsigned int c )
{
  printf( "%u\n", c );
  image_thrash_by_pointer( I, c+1 );
}
```

De här funktionerna gör alltså ingenting vettigt, utan bara anropar sig själva rekursivt och skriver ut en räknare.

Modifiera funktionerna ovan så att de förutom `c` också skriver ut bredd och höjd på den `image` som kommer in. Skapa sedan en variabel av din typ `image`, fyll i ett värde för bredd och höjd (t ex 500 och 500), och anropa sedan `image_thrash_by_value( I, 0 )` respektive `image_thrash_by_pointer( &I, 0 )`.

[4] I alla fall så länge vi inte använder dynamisk minneshantering.

Fråga 6

Båda funktionerna kommer att skriva ut samma värden. Däremot kommer ditt program att krascha så småningom, i båda fallen. Vilken version kör längst? Varför? (*Tips:* Det har att göra med storleken på anropsstacken.)

2.9 Bildbehandling

Nu ska du skriva ett program som läser in en bildfil, till exempel den i fig. 1a.

Du kommer att läsa och skriva bilder på formatet PGM, som är ett väldigt enkelt bildformat.[5] PGM-filer är textfiler som börjar med strängen ”P2”, sedan två heltal som anger bredd och höjd på bilden, efter det ett tal som anger maxvärdet för en pixel (om maxvärdet är 255 betyder det att svarta pixlar har värdet 0 och vita har värdet 255). Efter dessa värden (som kallas *header*) anges bildens pixlar, med en siffra per pixel, som anger hur ljus den är. En PGM-fil kan också innehålla kommentarer, som börjar med `#`, men det hoppar vi över i den här uppgiften. Öppna filen `garvis.pgm` i en texteditor så ser du hur filen ser ut. Du kan också dubbelklicka på `garvis.pgm` i filhanteraren så ser du hur bilden ser ut. Det finns också två små bilder som är med som exempel, och som är bra att testa din kod på: `gradient.pgm` och `hash.pgm`. De är bara 5×5 pixlar stora. Kolla på dem också i en texteditor och med bildvisaren från filhanteraren.

Första uppgiften är att läsa in en PGM-fil. Ditt program ska ta ett filnamn som in-argument, och läsa in den filen (förutsatt att den finns).

Använd din datastruktur `image` från den föregående uppgiften. Skapa en variabel med typen `FILE*` för att läsa in filen. Använd `fopen` och `fscanf` för att läsa in den och fylla i en `image`-variabel. Se till att sluta läsa när med `fscanf` när filen tar slut. Skriv ut, med `printf`, värdet på den första och sista pixeln (på samma rad med mellanslag emellan), för att kontrollera att du har läst in rätt värden.

Glöm inte att stänga filen med `fclose` när du är färdig.

Exempel:

```
./task_9 garvis.pgm
160 9
```

2.10 Tröskling

Nu ska du (äntligen) skriva ett litet bildbehandlingsprogram som *trösklar* en bild, så att den ser ut som i fig. 1b. Operationen går helt enkelt till så att alla pixlar som har ett värde mindre än ett visst tröskelvärde sätts till 0, och alla andra pixlar sätts till maxvärdet (255). Figur 2 visar ett exempel på testbilden `gradient.pgm` före och efter tröskling, med pixelvärden.

Nu behöver ditt program skapa en ny `FILE*` och öppna den med `fopen` för *skrivning*. Gå sedan igenom din bild-datastruktur och skriv ut en korrekt PGM-fil som är trösklad på värdet 140. (Pixlar som har värdet 140 och över sätts till 255, och övriga sätts till 0.) Skriv den nya bilden till `threshold.pgm`. Läs in filnamnet från kommandoraden som i föregående uppgift.

Din tröskling ska utföras i en egen funktion, och inte i `main`. Du ska alltså skriva en funktion som tar in en bild enligt din datastruktur, och skickar ut en trösklad version av samma bild. Den här funktionen ska inte skriva till fil, utan det gör du sedan i `main` (eller, ännu snyggare, i en separat funktion som bara har som syfte att skriva bildfiler).

Du ska *inte* använda en global variabel för din bild. Du måste skapa den i `main` och skicka den som argument till dina övriga funktioner.

[5] Se `http://netpbm.sourceforge.net/doc/pgm.html`

(a) Original

(b) Trösklad

(c) Suddig

Figur 1: Dansbandet Garvis i olika versioner.

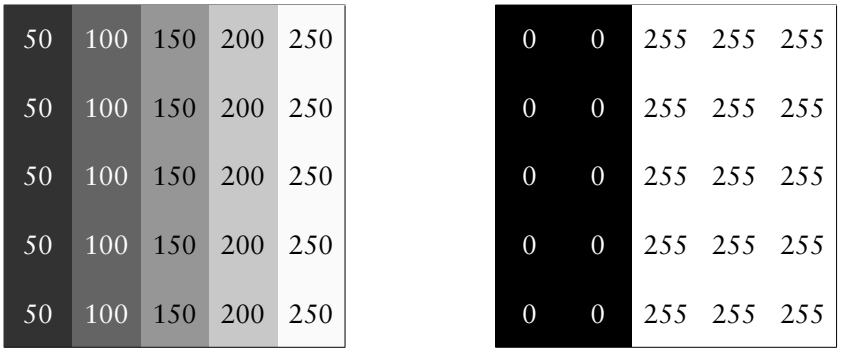
I din funktion, väljer du att skicka pekare till `image` som in- och ut-argument, eller väljer du att returnera en `image`? Vilket tycker du är att föredra, och varför?

2.11 Sudda

Slutligen ska du skriva ett program som gör bilden suddig, som i fig. 1c. Det här är lite knepigare, eftersom du måste jämföra pixlar som ligger bredvid varandra. Ut-värdet på varje pixel i du skriver ska nu vara medelvärdet av pixlarna som ligger till vänster, höger, ovanför och nedanför i, samt i själv. Då behöver du leta på rätt ställe i fältet där du har lagrat in-pixelvärdena. Om du har ett endimensionellt fält kommer då pixeln på raden under i ligga på plats $i + w$, där w är bildens bredd.

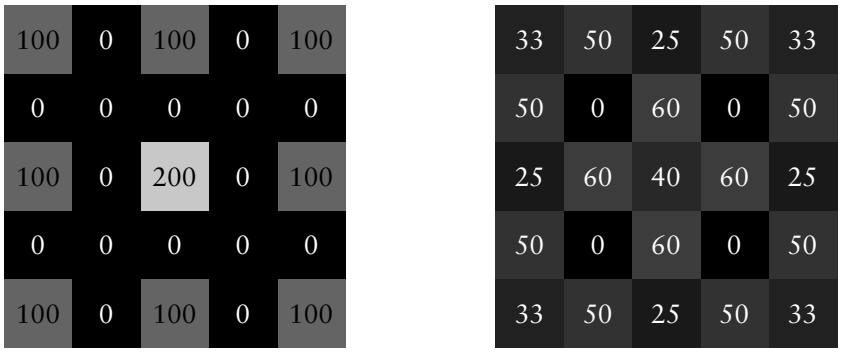
Viktigt här är att se till att du inte försöker läsa på en plats som ligger utanför fältet, vilket är lätt hänt, framför allt på första och sista raden i bilden! (*Tips:* debug-verktyget i Geany är användbart här, för att bena ut vad som händer om det inte blir som du tänker dig!)

Du behöver hantera första och sista raden lite annorlunda än de andra. Det finns olika strategier för det här, men i den här uppgiften ska du hoppa över du värden som inte finns. Med andra ord ska du skriva medelvärdet av (vänster + höger + upp) för den nedersta raden, t ex. Även den vänstra och högra pixelkolumnen behöver hanteras separat, jämfört med pixlarna mitt i bilden. Gör på samma sätt här som



(a) Original (b) Trösklad

Figur 2: Testbilden gradient.pgm



(a) Original (b) Suddig

Figur 3: Testbilden hash.pgm

med den översta och understa raden i bilden. Med andra ord ska du inte bara läsa vidare och ta en pixel från föregående eller nedanstående rad.

Figur 3 visar testfilen `hash.pgm` före och efter suddning. Du kan ha de här värdena som referens när du gör ditt program. Mittenpixeln i den suddade bilden har alltså värdet $(200 + 0 + 0 + 0 + 0)/5 = 40$, och hörnpixlarna är $(100 + 0 + 0)/3 = 33$.

Suddningsfunktionaliteten ska ligga i en egen funktion, och inte i `main`, precis som i föregående uppgift. Du får heller inte använda en global variabel för din bild, utan måste skicka den som argument till dina övriga funktioner.

Skriv den suddiga bilden till `blur.pgm`. Den ska ha samma antal pixlar som input-bilden.

2.12 Binära talrepresentationer

Fråga 7 I din rapport, beskriv hur man representerar följande tal i binär form, som en **`unsigned char`** (alltså *utan* tvåkomplement). Något av talen går inte att representera på det här sättet. Ange vilket, och varför.

Fråga 8 Beskriv hur man representerar talen i binär form, som en **`signed char`** (alltså *med* tvåkomplement). Något av talen går inte att representera på det här sättet. Ange vilket, och varför.

- 0
- 8
- 10
- 130
- −126